Block Copy Using Pentium® lll Streaming SIMD Extensions
Revision 1.9

January 12, 1999

Table of Contents

1 Block Copy Overview

One of the important capabilities of a CPU that impacts performance is its ability to efficiently move blocks of memory from one place to another. Much effort has been made by software developers in this area to make this operation more efficient. This paper shows how block copy speed can be dramatically improved by using new instructions available on the Pentium III processor to do streaming stores to memory. Streaming stores bypass system caches when copying single-use data to memory to enable the CPU to efficiently utilize the system bus bandwidth, significantly improving the speed of block copies.

1.1 Streaming SIMD Extensions

In addition to a wealth of new instructions directed toward multi-media applications, the Pentium III processor provides four instructions, **prefetch**, **movntps, movntq** and **movups**, designed specifically to improve processor performance when accessing memory.

1.1.1 prefetch

This instruction enables you to fetch data from memory and place it into the specified memory cache hierarchy. The benefit of this is that single-use data, such as the source for a memory copy, can be placed directly into the L1 cache, bypassing any other memory caches in the system. This prevents multiple-use data in the memory caches from being evicted by single-use data. Also, the CPU doesn't have to wait for the memory data to percolate through the memory caches, it can be fetched directly into the L1 cache. Another advantage is that prefetch is a non-blocking instruction and can be executed out of order without any dependencies on other instructions. This has the effect of allowing the CPU to overlap the memory access with the execution of other instructions, effectively reducing the memory latency.

1.1.2 movntps

This instruction enables the CPU to store 16 bytes of data from a SIMD floating-point register to memory without requring a write allocate on any memory caches in the system. This instruction maintains proper cache consistency for data currently in the cache, either by evicting that data from the cache or by replacing it.

1.1.3 movntq

This instruction opeates the same as **movntps** except that it uses the 64-bit MMX™ registers and only operates on 64 bits of data at a time. It still copies data to memory without doing a write allocate on any memory caches in the system.

1.1.4 movups

This instruction moves 16 bytes of data to or from a SIMD floating-point register. Although this instruction searches the normal cache hierarchy to find the memory data, it has the advantage that it operates on 16 bytes of data at a time but does not suffer from the restriction that the memory address must be 16-byte aligned.

1.1.5 sfence

This instruction forces the write combining (WC) buffers to be written to memory. Because the Streaming SIMD Extensions store instructions use the WC buffers, **sfence** is needed to ensure memory coherence for streaming stores.

2 Copy using Streaming SIMD Extensions

When using the Streaming SIMD Extensions the following is the basic algorithm for an efficient streaming memory copy:

1. Prefetch the data into the L1 cache.
2. Move the data into a SIMD floating-point register with **movups**.
3. Write the data into memory with **movntps**.
4. Repeat the preceding two steps as needed to complete the copy.

Given this basic algorithm there are a few issues that need to be solved before this algorithm can be used, especially inside an OS.

- The alignment of the data can affect the efficiency of the copy.
- The source addresses must be in the Translation Lookaside Buffer (TLB) for the prefetch to work.
- All of the source bytes need to be prefetched into the L1 cache before the copy can be done.
- The interior copy loop should be unrolled but there are limitations that will be described in more detail later.
- Using the SIMD floating-point registers modifies a part of the floating point state of the CPU and this modified state needs to be saved and restored.
- The **sfence** instruction is needed to serialize memory accesses.
- There are some caveats that could cause the streaming copy algorithm to actually execute slower than a simple **rep movsb** loop.

All of these issues will be examined in detail in the remainder of this section.

2.1 16-byte alignment

The **movntps** instruction that is used by this algorithm requires memory addresses that are 16-byte aligned. This raises two issues that have to be resolved:

1. The destination address needs to be 16-byte aligned. This is achieved by putting in a **rep movsb** loop at the beginning of the routine that copies from 1 to 15 bytes, as many as are needed to get the destination aligned on a 16-byte boundary.
2. There could be from 1 to 15 bytes remaining after the last 16-byte copy. The real copy routines resolves this problem with a simple **rep movsb** as needed at the end of the routine.

Note that the algorithm has no alignment constraints on the source data since the **movups** instruction used to read the source bytes from memory has no alignment constraints.

2.2 TLB priming

The **prefetch** instruction works only if the data address is already in the TLB. If the address is not in the TLB then the **prefetch** instruction becomes a simple no-op and has no effect on any of the system caches. To resolve this, two extra memory references are added to the beginning of the copy loop, one to reference the first byte of the source buffer and the second to reference the last byte of the buffer. Note that two memory references are required since the buffer could be split across a 4K page boundary which would mean that the buffer was mapped by two different TLB entries. Prefetching the first byte of the buffer get the first entry into the TLB and prefetching the last byte of the buffer gets the second entry into the TLB.

This assumes that the buffer is no bigger than 4K in size which is not a problem since the copy routine presented here expects to be called for no more than a 4K copy. This is the typical page size for the Pentium III processor and therefore there should be very few instances where the OS tries to copy more than 4K bytes at a time. If the OS expects to do lots of large copies then **bcopy** should be modified to break the copy up into 4K chunks and use the algorithm presented here for each 4K chunk.

2.3 Prefetch all source bytes

Any load instruction, including **prefetch**, that misses the cache must contend for the same CPU resources used by the **movntps** store instruction. This causes premature evictions from the cache and partial bus transactions, both of which slow down the copy operation. To solve this problem, prefetch all the source bytes first and then do the copy. Since the Pentium III processor has a 16K L1 data cache and the maximum buffer size dealt with by this algorithm is 4K this is a perfectly viable solution.

2.3.1 Pre-emption

Operating systems that allow for pre-emption create a potential problem. If the system pre-empts the **bcopy** after the data is prefetched but before the copy occurs, all the benefits of the prefetch can be lost in the process of switching to a different context and back again. For most operating system code this is not a problem. Inside the OS, pre-emption normally occurs at specific points and it is easy to protect the **bcopy** code from pre-emption. A user process however may get pre-empted at any time but hopefully this will not occur during a **bcopy**.

2.3.2 Prefetch Only

Another possibility is to increase the speed of the regular **bcopy** routine by just prefetching all its source bytes first.. This technique yielded very disappointing results. This yielded a copy speed of 260 MBytes/sec vs. a normal **bcopy** speed of 255 MBytes/sec, a very minimal 2% gain. This technique potentially improves cache performance by not evicting unnecessary data from the cache but it has very minimal impact on **bcopy** speed.

2.4 Loop unrolling

Since there are 8 SIMD floating-point registers there is a temptation to unroll the main loop of the **bcopy** routine and do 8 reads and writes per loop rather than just 1. As it turns out, there is a significant performance boost in unrolling the loop to do 2 reads and writes. This speeds the maximum copy speed from about 570 MBytes/sec to 634 MBytes/sec. Unrolling the loop from 2 reads and writes to 8 reads and writes only speeds the copy up by about 0.5%, to a maximum of 637MBytes/sec. Given the overhead of saving 6 extra SIMD floating-point registers, whose overhead is added to all copies, the performance boost from full unrolling is not worth the effort. The **bcopy** algorithm presented here only unrolls the loop twice.[1]

2.5 Floating Point (FP) State

The programmer should note that the SIMD floating-point registers that are used in this algorithm are part of the floating point state. Typically the OS strictly uses integer code and does not deal with saving the FP state or the SIMD floating-point state while executing inside of the kernel. This algorithm, by using the SIMD floating-point registers, violates that assumption.

---

[1] Experiments have also shown that unrolling the loop from 1 to 8 registers increases the partial register writes on the bus from about 8,000/sec. up to 1,500,000/sec., a situation that can degrade total system performance.

One possible solution to this problem is to use the **FXSAVE**/**FXRSTOR** instructions to save and restore the FP state whenever the kernel does a block copy. This is not a good idea for two reasons:

1. The **FXSAVE** save area is 512 bytes. The **FXSAVE/FXRSTOR** instructions copy 512 bytes to memory for the save and read 512 bytes from memory for the restore. Even though the **FXSAVE** instruction does a streaming store it still takes a significant amount of time to copy 512 bytes of data to and from memory.
2. Lazy FP state save. Most OS's do not save the floating point state or the SIMD floating-point state on every context switch. Instead they use the fact that if the **CR0.TS** bit is set, the CPU generates a Device Not Available (DNA) fault if a user process attempts to execute a floating point or SIMD floating-point instruction. The kernel sets the **CR0.TS** bit on every context switch, causing the current process to generate a DNA fault the first time it tries to use the floating point or SIMD floating-point unit.[2] The DNA fault handler can determine if the current process was the last process to use one of the floating point units or if a different process last used the floating point units. In the first case it clears the **CR0.TS** bit and returns; in the latter case it saves the floating point state for the other process, restores the floating point state for the current process if necessary and finally clears the **CR0.TS** bit and returns. This becomes very problematic for an OS that wants to use the SIMD floating-point unit since a process that is using the floating point unit has already cleared the **CR0.TS** bit and therefore the kernel will not generate a DNA when it uses a SIMD floating-point register. Modifying the kernel to set the **CR0.TS** bit appropriately is very difficult , could involve many changes and can result in many more DNA faults.

Fortunately, there is a simple solution to this problem that does not involve saving the complete FP state or taking DNA faults. Instead, the **bcopy** routine can just:

1. Get the current value of **CR0**.
2. Clear the **CR0.TS** bit so that a DNA will not occur.
3. Save the SIMD floating-point registers the **bcopy** will use.
4. Execute the copy.
5. Restore the SIMD floating-point registers that were used.
6. Restore the original value of **CR0**.

Clearing the **CR0.TS** bit prevents a DNA from occuring and saves only the SIMD floating-point registers used by the copy routine, which results in writing 32 bytes to memory rather than 512.

2.5.1 DNA Faults

The algorithm presented here can also be used by user-level code. In this situation there is no need for the block copy code to save and restore the SIMD floating-point registers, these registers will be saved as necessary by the OS. Note that these DNA faults will have an impact on the user's block copy speed, as detailed in section 3.2.

[2] There is an obvious optimization that all OS's apply which is to remember which process last used the floating point unit and then the OS clears the **CR0.TS** bit whenever it context switches to that process. This avoids the DNA fault for the common case of a single floating point process that gets switched out momentarily in favor of an integer only process.

2.6 sfence Instruction

The **movntps** instruction uses the Write Combining (WC) buffers to buffer data to memory. The WC buffers are not flushed to memory unless new data needs to be written to the buffers. Therefore, the **bcopy** routine must flush these buffers to memory before it completes or else a normal write to memory would bypass this buffer and write stale data to memory. The **sfence** instruction provides the serialization required to guarantee that all data has been completely written out to memory.[3]

2.7 Caveats

Note that there are situations where using the Streaming SIMD Extensions algorithm presented in this paper could result in code that actually runs slower. This could occur when the destination data is re-used immediately after the copy. In this situation having the Streaming SIMD Extensions copy code bypass the system data will result in 2 accesses to memory rather than 1 - one to do the copy and one to load the data into the data cache for the second use. This might happen if the Streaming SIMD Extensions copy is used to initialize a data structure immediately before that data structure is used.

3 Benchmark results

A benchmark program was created that attempts to simulate the copy operations that an OS would typically perform. The benchmark program first allocates a 4M input buffer and a 4M output buffer. Then, for the copy size specified, the program copies successive blocks from the input buffer to the output buffer. When the program reaches the end of the buffers it cycles back to the beginning of each buffer. By doing successive copies over a 4M range the program attempts to force each copy to operate on data that is not inside any of the memory caches.

The benchmark program does the copy operations for a specified number of seconds. At the end of the run the program calculates the number of megabytes per second of data transferred over the system bus during the run, counting 2 bytes transferred over the bus for every byte copied (1 transfer to read the byte and 1 transfer to write the byte).

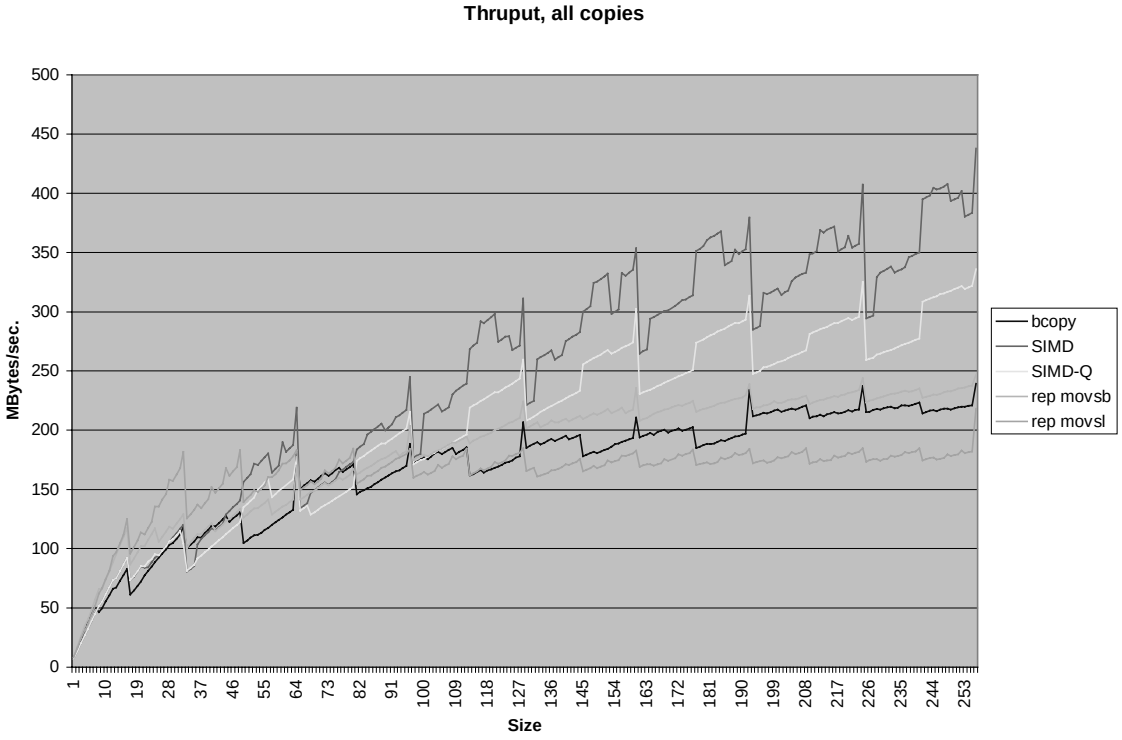
3.1 Speed of multiple copy algorithms in the OS

The first study done was to determine which of the following five algorithms produced the fastest copy speed:

1. **The standard system *libc bcopy* routine**. This routine is 254 bytes of highly optimized code that takes into account the size of the copy and the alignment of the source and destination and then does either **rep movsl**, **rep movsb** or register copies. The data for this algorithm is labeled **bcopy** on the graph.
2. **Simple *rep movsb* copy**. This routine just does the copy with one repeated move byte instruction. The data for this algorithm is labeled **rep movsb** on the graph.

[3] Early versions of the Pentium® III processor had an errata where a streaming store followed by an APIC write caused the APIC write data to be corrupted. This errata can be avoided by placing a serializing instruction after the **sfence** instruction. This errata has been fixed in B0 and later steppings of the Pentium III processor.

3. **Simple *rep movsl* copy**[4]. This routine does the copy in two phases, first copying as many bytes as possible with a repeated move 4-byte word instruction and then copying the remaining 1-3 bytes with a single **rep movsb**. The data for this algorithm is labeled **rep movsl** on the graph.
4. **Streaming SIMD Extensions using 64-bit MMX registers**. This is the Streaming SIMD Extensions algorithm presented in this paper except 64-bit MMX registers were used instead of 128 bit SIMD floating-point registers. The data for this algorithm is labeled **SIMD** on the graph.
5. **Streaming SIMD Extensions using 128-bit SIMD floating-point registers**. This is the Streaming SIMD Extensions algorithm presented in this paper. The data for this algorithm is labeled **SIMD** on the graph.

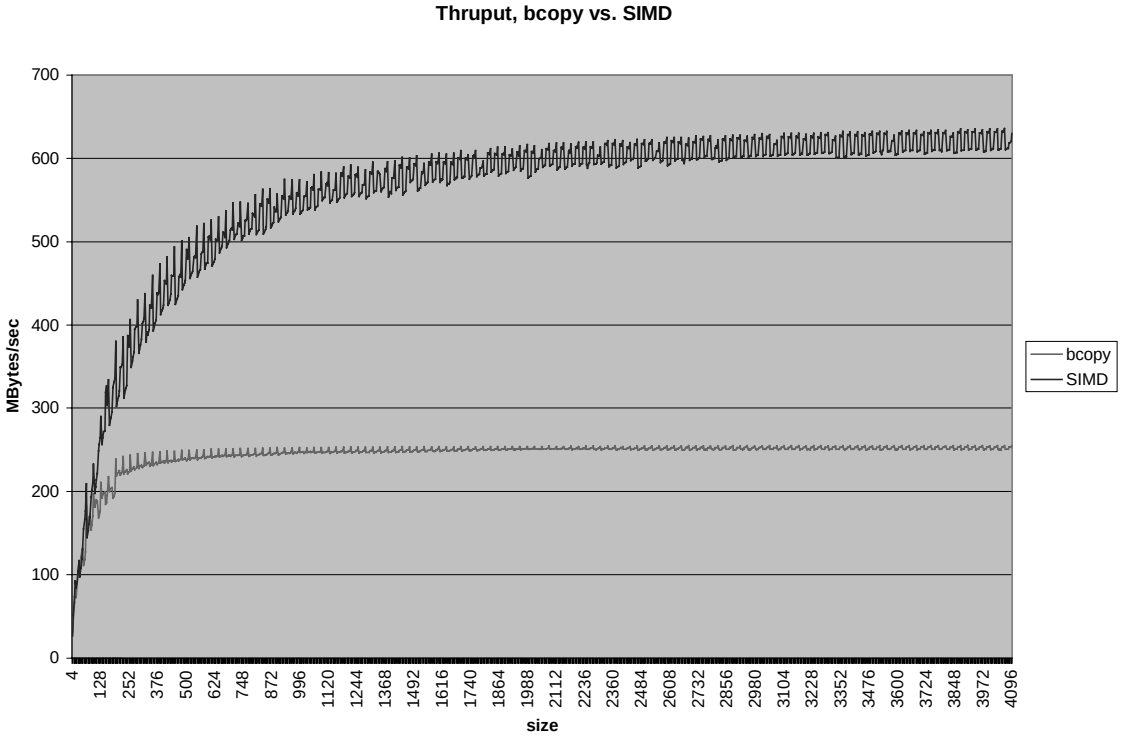


Some interesting points are brought out by this graph:

1. Surprisingly enough, the simple **rep movsb**, in general, is the fastest of the traditional copy algorithms.
2. The **rep movsl** is the fastest algorithm for small block sizes of 48 bytes or less. After that the 128-bit Streaming SIMD Extensions algorithm starts to speed up dramatically.
3. The SIMD floating-point algorithm demonstrates a noticeable performance advantage over the 64-bit MMX algorithm. This is an interesting result since this particular benchmark should be limited by the bus bandwidth, not the MMX registers used to access the data. This could be an anomaly related to the motherboard on the Seattle machine used for these benchmarks.

[4] The Linux* 2.0 kernel uses an assembler inline function to implement block copies. The inlined function turns out to be almost exactly the same as the simple string move routine presented here and the performance of this inlined function exactly matches the performance of this routine.

Extensive runs were done to see how fast the Streaming SIMD Extensions **bcopy** was in relation to the standard Linux* 2.0 **bcopy** routine. The results of these runs are presented in the following two graphs.



This graph shows the speed of the Streaming SIMD Extensions **bcopy** versus the system **bcopy** for all 4 byte block sizes ranging from 4 bytes to 4096 bytes. As can be seen the Streaming SIMD Extensions **bcopy** is clearly superior, once the block size becomes greater than about 128 bytes. The Streaming SIMD Extensions **bcopy** quickly rises in speed until it asymptotically approaches a maximum rate of about 635MBytes/sec while the standard system **bcopy** routine only approaches a maximum rate of about 255 MBytes/sec. The fact that the system **bcopy** is faster for small sizes is expected since there is a certain amount of overhead associated with setting up the Streaming SIMD Extensions copy.

This graph shows the speed of the Streaming SIMD Extensions **bcopy** versus a simple **rep movsb** for every block size from 1 to 128 bytes. As can be seen, the system **bcopy** is marginally faster for block sizes up to 44 bytes but when the block size is greater than 44 bytes the Streaming SIMD Extensions **bcopy** routine is clearly faster. And, as shown in the first graph, the performance gap gets significantly wider as the block size increases.

3.1.1 System Impact

The studies done so far, as represented by the previous graphs, show how fast a simple copy operation can be speeded up using the Streaming SIMD Extensions. Given that the Streaming SIMD Extensions copy only shows a benefit for copies that are larger than 44 bytes the next issue is to find out how many copy operations of what size the kernel does. To measure these numbers the block copy routines inside of the Linux* 2.0 kernel were instrumented. The instrumentation used was to add 13 bucket counters to the system, one for each power of 2 block size from 1 to 4096, and a 14th bucket for all block sizes greater than 4096. As it turns out, the Linux* 2.0 kernel uses two different copy routines, one for copies where the size is known at compile time and another for variable length copies. For this reason there were actually two sets of buckets maintained, one for variable sized copies and one for fixed size copies. The copy routines were then modified to find the next power of two that was greater than the copy size requested and the copy routine then incremented that bucket counter.

The benchmark chosen was to simply re-compile the Linux* 2.0 kernel. This was a reasonable, simple test that would exercise most of the file system and memory management areas of the kernel. A second benchmark was run, also just a re-compile of the kernel, but this time the entire source tree was maintained over a 10 Mbit TCP/IP network on a different server machine. This second benchmark would show the copy behavior of the network stack.

The primary information from these two benchmarks was the number of seconds of user time, system time and elapsed time it took to complete the benchmark, represented in the following table.

	Local	NFS
User	253.95	256.42
System	14.71	17.95
Elapsed	280.22	549.06

The results of these two benchmarks are summarized in the following table:

Copies to rebuild the kernel					
	Local			NFS	
Size	Variable	Fixed		Variable	Fixed
1	583	0		6,076	0
2	76	0		609	0
4	535	0		3,868	59
8	1,158	33		158,035	67,263
16	975	19		20,318	30,109
32	685	0		50,760	0
64	153	0		1,040	0
128	167	0		16,649	0
256	210	20		30,029	550
512	234	2,773		1,044	3,285
1024	4,325	15		2,112	16
2048	0	1,723		38,375	1,934
4096	0	32,239		0	35,932
> 4906	0	0		0	0

The real metric of interest is the number of bytes transferred. Given that the speed of the Streaming SIMD Extensions copy is known, this makes it possible to determine how much improvement the Streaming SIMD Extensions copy would provide for this benchmark. Assuming, as an upper bound, that each copy was the maximum size indicated by the bucket then multiplying the bucket size by the bucket could give the number of kilobytes transferred. The following table gives those numbers:

Appendix C gives a table that shows the usage of block copy operations inside the kernel during two different builds of the Linux* 2.0 kernel. One of the builds operated on a source tree that was located on a local drive while the other build used an NFS mounted source tree. Given that the benefit of the Streaming SIMD Extensions copy only applies to copies greater than 64 bytes it is fortunate that the bulk of the copies in this experiment fall into this category. Unfortunately there just aren't enough copies being done by the kernel to cause an appreciable performance improvement by using the Streaming SIMD Extensions copy. As the data in Appendix C shows, for the local benchmark there were 138 MBytes that would have taken, at 255 MBytes/sec, 0.54 seconds using a normal copy and would have taken, at 635 MBytes/sec, 0.22 seconds using a Streaming SIMD Extensions copy. This would reduce the System CPU time by 0.32 seconds or 2.18%. Unfortunately, this would only have improved the elapsed time by 0.11%. The results for the network benchmark were a little better, the system CPU would have taken 0.93 seconds using a normal copy and would have taken 0.38 seconds using a Streaming SIMD Extensions copy. This would have reduced the system CPU time by .55 seconds or 3.1% while the elapsed time would have improved by 0.11%. These results are illustrated in the following table:

System time	Time	SIMD improvement	SIMD Time	Delta
Local	14.71	0.32	14.39	2.18%
Remote	17.95	0.55	17.40	3.10%
Elapsed time				
Local	280.22	0.32	279.90	0.11%
Remote	489.06	0.55	488.51	0.11%

3.2 Speed of user level block copy

Benchmarks were also run in user mode to see what effect the overhead of taking DNA faults would have on the Streaming SIMD Extensions block copy speed. The basic benchmarking technique used was the same technique that was used to measure the speed of block copies inside the kernel. The only difference is that there was no saving and restoring of the SIMD floating-point registers used by the block copy algorithm. These registers are simply part of the user's FP context and the kernel saved and restored them appropriately, based upon any DNA faults that occurred.

Benchmarks of user level code were run under Windows* 98 and Windows NT* 4.. For Windows NT OS's the benchmarks were run on both single-processor and dual-processor machines. Also, all benchmarks were run using both a cold cache (the source data was guaranteed not to be in the cache) and a warm cache (the source data was guaranteed to be in the cache). The graphs in Appendix D detail the results of these experiments.

The major goal of these benchmarks was to discover the block size at which the block copy using the Streaming SIMD Extensions algorithm became faster then the **rep movsl** algorithm that is part of the standard Microsoft C library.

UP and MP cold cache copying crosses over (becomes faster using the Streaming SIMD Extensions algorithm) between 1K and 2K block sizes. UP and MP warm cache copying crosses over at just below an 8K block size.

The basic overhead of the DNA during Streaming SIMD Extensions startup is about 1000 clocks for the Windows NT 4.0 MP warm cache case, with about 350 more clocks for the UP case. The Windows NT 4.0 MP cold cache case comes in at 4800 clocks, with about 2000 more clocks for UP. The numbers for Windows 98 show more variation but are similar to the Windows NT 4.0 UP numbers.

Pre-emption was not found to have a significant effect on the crossover points (it did widen the standard deviations significantly). The code currently makes no attempt to force or prevent context switching during the block copy algorithm.

4 Conclusion

This paper shows how to use the Streaming SIMD Extensions to increase the block copy speed inside an OS kernel from a maximum of 255 MBytes/sec up to 635MBytes/sec, an improvement of 149%. This increase can be achieved with minimal change to the OS and with no change to the floating point save and restore mechanism.

This dramatic improvement has to be balanced by the fact that, for a real-world application running on Linux* 2.0, this change has only been shown to improve the system CPU usage by approximately 2%.

Appendix A. The Streaming SIMD Extensions bcopy routine, C language version

```
#define PREFETCHNTA(ad) __asm__("prefetchnta %0" : : "m" (*(ad)))
#define MOVUPS(ad,reg)  __asm__("movups %0,%%"#reg"\n" :: "m" (*(ad)));
#define MOVNTPS(reg,ad) __asm__("movntps %%"#reg",%0\n" :"=m" (*(ad)));

/*
 *  copy_simd - copy using Streaming SIMD Extensions
 *
 *  src  - source buffer
 *  dst  - destination buffer
 *  size - number of bytes to copy
 */
      void
copy_simd(src, dst, size)
char *src, *dst;
int size;
{
      char *pre, *svp;
      char xmm_save[3 * 16];
      int i, cr0;

      if (size >= 32) {
#ifdef      KERNEL
            __asm__("movl     %%cr0,%0\n\t"
                  "clts\n\t"
                  : "=r" (cr0));
            svp = (char*)((long)(xmm_save + 15) & ~0xf);
            MOVNTPS(mm0, svp);
            MOVNTPS(mm1, svp + 16);
#endif      // KERNEL
            __asm__("movb %0,%%eax\n\t"
                  "movb %1,%%eax\n\t"
                  :
                  : "m" (*src), "m" (*(src + size - 4))
                  : "eax");
            pre = (char *)((long)src & ~0x1f);
            for (i = 0; i < size; i += 32)
                  PREFETCHNTA(pre + i);
            if ((i = ((int)dst & 0xf))) {
                  i = 16 - i;
                  __asm__("cld\n\t"
                        "rep; movsb"
                        :
                        : "S" ((long)src), "D" ((long)dst), "c" (i)
                        : "si", "di", "cx", "memory");
                  size -= i;
                  dst += i;
                  src += i;
            }
            while (size >= 32) {
```

```
                MOVUPS(src, mm0);
                MOVUPS(src + 16, mm1);
                MOVNTPS(mm0, dst);
                MOVNTPS(mm1, dst + 16);
                size -= 32;
                src += 32;
                dst += 32;
          }
#ifdef    KERNEL
          MOVUPS(svp, mm0);
          MOVUPS(svp + 16, mm1);
          __asm__("movl     %0,%%cr0\n\t"
                :
                : "r" (cr0));
#endif    // KERNEL
    }
    if (size)
          __asm__("movl %0,%%esi\n\t"
                "movl %1,%%edi\n\t"
                "movl %2,%%ecx\n\t"
                "rep; movsb"
                :
                : "m" (src), "m" (dst), "r" (size)
                : "esi", "edi", "ecx");
    __asm__("sfence");
    return;
}
```

Appendix B. The Streaming SIMD Extensions bcopy routine, assembly language version

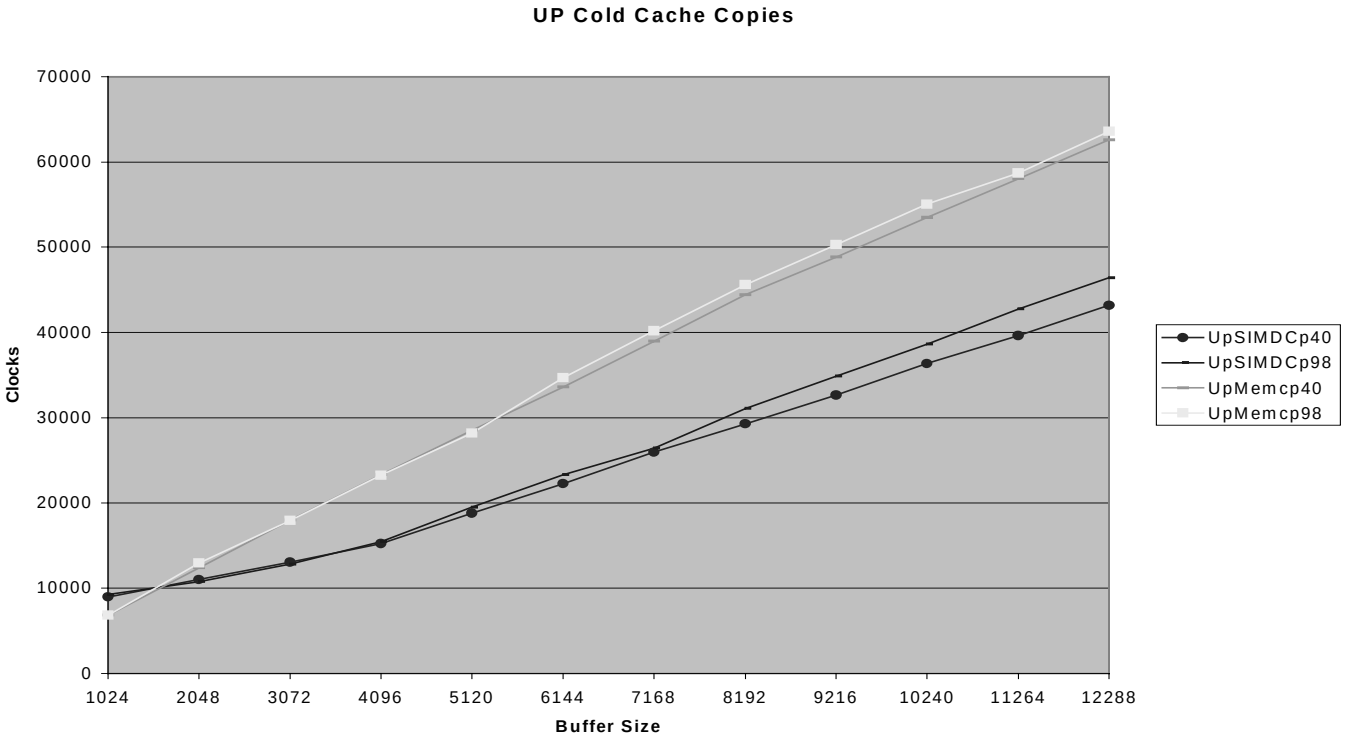
```
      .align 4
.globl bcopy_simd
      .type  bcopy_simd,@function
bcopy_simd:
      pushl %ebp
      movl %esp,%ebp
      pushl %edi
      pushl %esi

      movl 16(%ebp),%ecx      # size
      movl 8(%ebp),%eax       # source
      orl 12(%ebp),%eax       # destination

      cmpl $31,%ecx
      jle bcopy4              # size < 32

      movl 8(%ebp),%edx
      movb (%edx),%eax        # touch first byte of src
      movb -4(%ecx,%edx),%eax # touch last byte of src

      andb $0xe0,%dl          # align src ptr to 32-byte boundary
      xorl %eax,%eax

bcopy1:
      prefetchnta (%eax,%edx) # prefetch 32-bytes of source
      addl $32,%eax
      cmpl %ecx,%eax
      jl bcopy1

      movl 8(%ebp),%esi       # get src ptr
      movl 12(%ebp),%edi      # get dst ptr

      movl %edi,%eax          # get dst ptr
      andl $15,%eax           # check alignment
      je    bcopy2            # dst already 16-byte aligned
      movl %ecx,%edx          # save size
      movl $16,%ecx
      sub %eax,%ecx           # calculate alignment pad
      sub %ecx,%edx           # adjust size
      rep; movsb              # align destination
      movl %edx,%ecx          # new size
      jmp bcopy2a             # continue with copy

bcopy2:
      movups (%esi),%mm0      # get 16-bytes of source
      movups 16(%esi),%mm1    # next 16-bytes of source

      movntps %mm0,(%edi)     # store 16-bytes to destination
      movntps %mm1,16(%edi)   # store 16-bytes to destination
```

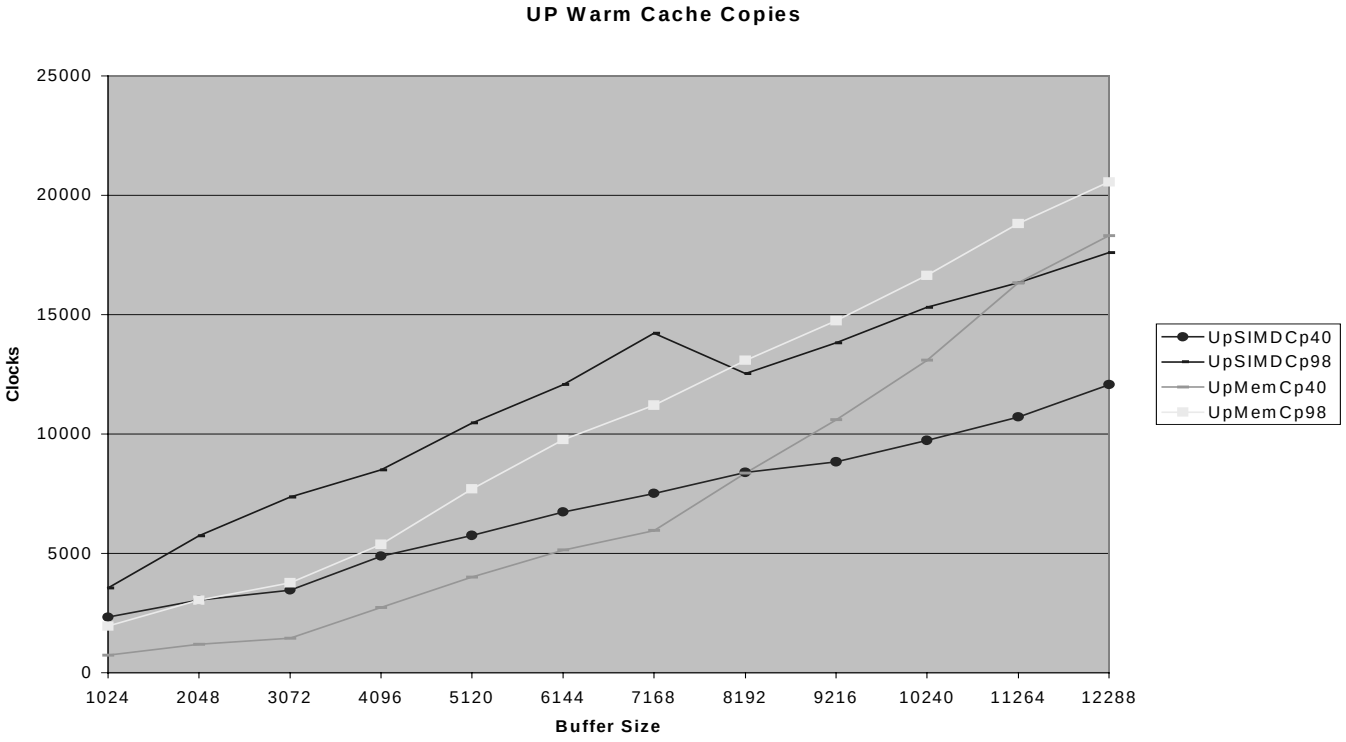
```
      addl $-32,%ecx          # decrement size
      addl $32,%esi           # increment src
      addl $32,%edi           # increment dst
bcopy2a:
      cmpl $31,%ecx
      jg bcopy2

bcopy4:
      testl %ecx,%ecx         # check if tail copy needed
      je bcopy5

      rep; movsb              # esi, edi & ecx already set up

bcopy5:
      sfence                  # flush the WC buffer

      popl %esi
      popl %edi
      popl %ebp
      ret
```

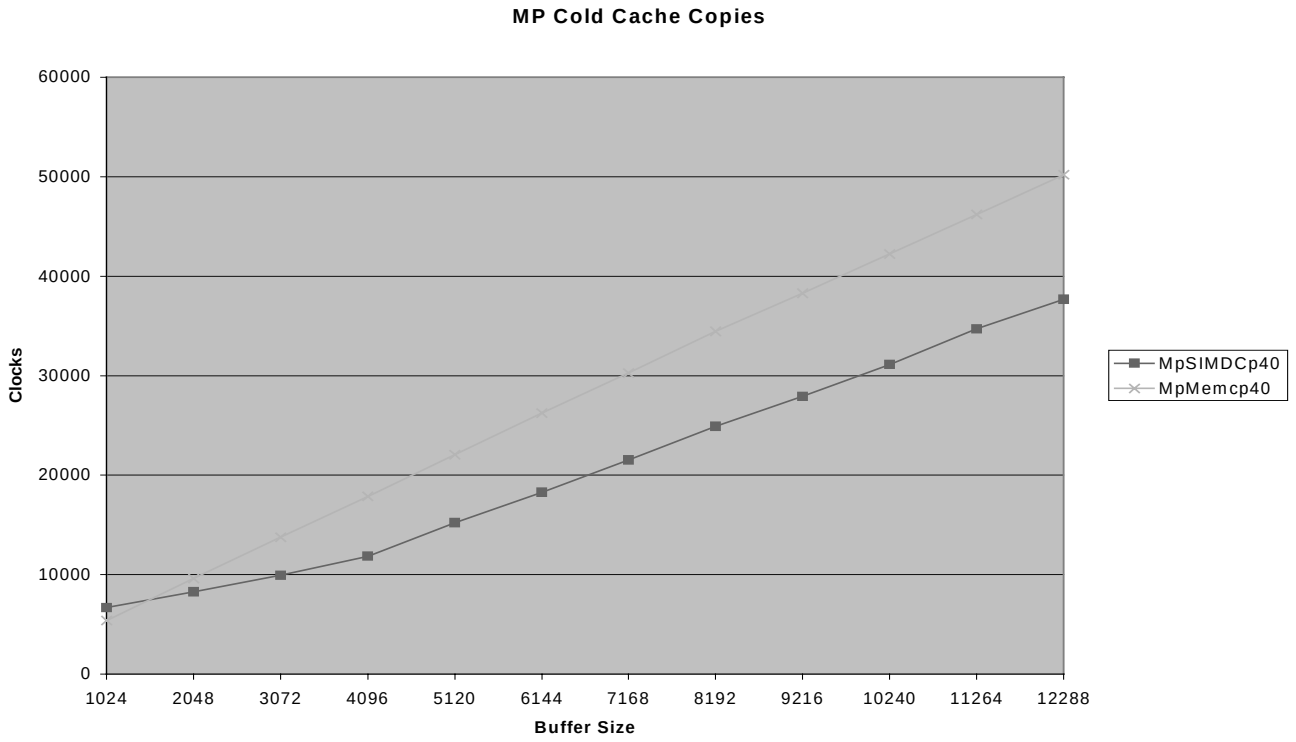
Appendix C. Block Copy Usage in the Linux* 2.0 kernel

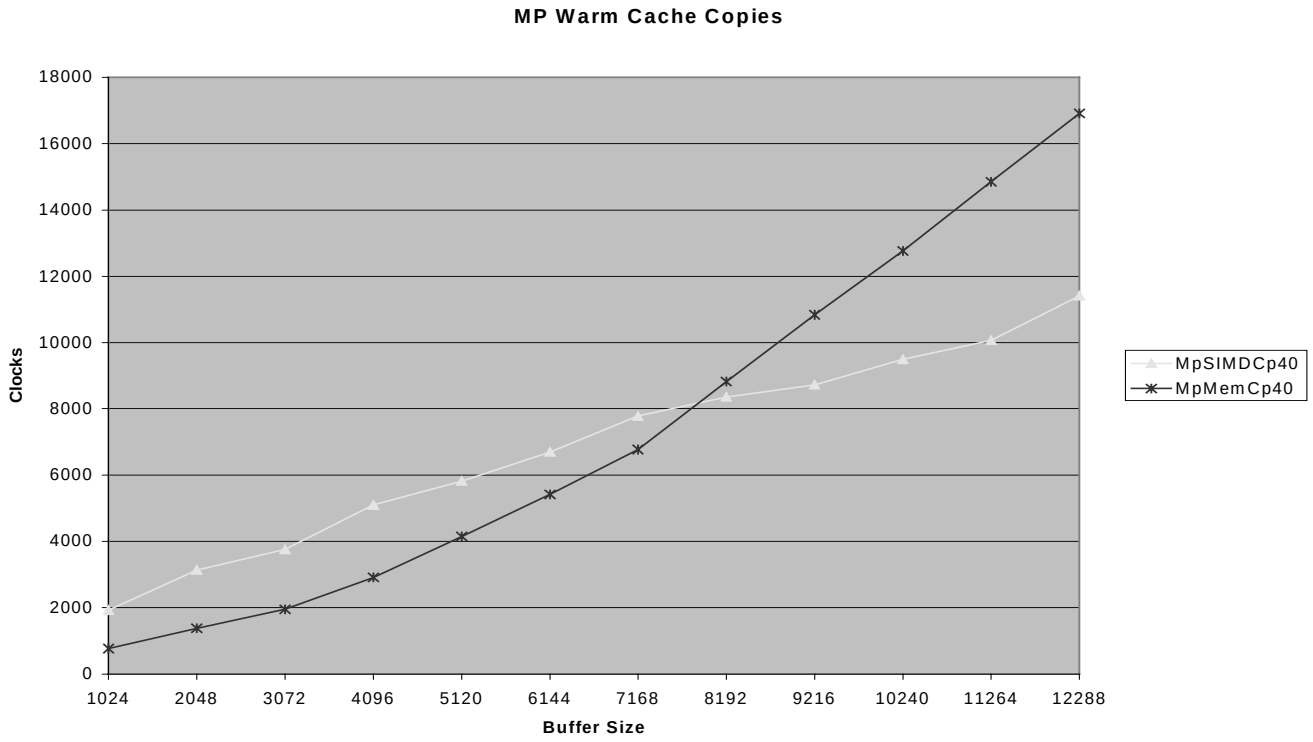
Kbytes Copied						
	Local		NFS			
Size	Variable	Fixed		Variable	Fixed	
1	0.57	0.00		5.93	0.00	
2	0.15	0.00		1.19	0.00	
4	2.09	0.00		15.11	0.23	
8	9.05	0.26		1,234.65	525.49	
16	15.23	0.30		317.47	470.45	
32	21.41	0.00		1,586.25	0.00	
64	9.56	0.00		65.00	0.00	
128	20.88	0.00		2,081.13	0.00	
256	52.50	5.00		7,507.25	137.50	
512	117.00	1,386.50		522.00	1,642.50	
1024	4,325.00	15.00		2,112.00	16.00	
2048	0.00	3,446.00		76,750.00	3,868.00	
4096	0.00	128,956.00		0.00	143,728.00	
> 4906	0.00	0.00		0.00	0.00	
			Total			Total
<= 64	48.50	0.55	49.05	3,160.60	996.18	4,156.78
> 64	4,524.94	133,808.50	138,333.44	89,037.38	149,392.00	238,429.38

Appendix D. User Mode Block Copy Benchmark Data






MP Cold Cache Copies
Clocks
60000
50000
40000
30000
20000
10000
0
1024
2048
3072
4096
5120
6144
7168
8192
9216
10240
11264
12288
Buffer Size
MpSIMDCp40
MpMemcp40


MP Warm Cache Copies
Clocks
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
1024
2048
3072
4096
5120
6144
7168
8192
9216
10240
11264
12288
Buffer Size
MpSIMDCp40
MpMemCp40